Storage Jamming

John McDermott David Goldschlag
Naval Research Laboratory

Introduction

In the past, the most likely motive for attacks that modify data would have been financial gain. The problem of fraud has been addressed by Clark and Wilson [2], by Sandhu and Jajodia [13], and by others [8, 9]. However, changes in technology have made many organizations so dependent on information systems that it is now possible to disrupt or degrade their operations by interfering with their supporting information systems [3]. When this disruption is accomplished by unauthorized modification of data we call it *storage jamming*.

Storage jamming is the malicious modification of stored data, done for the purpose of degrading or disrupting real-world operations that depend on the correctness of the data. We assume the person initiating the malicious modification (frequently via a Trojan horse) does not receive any direct benefit, financial or otherwise, but rather is motivated by more indirect goals such as improving the competitive position of his or her own organization. The target data need not be data stored by a general purpose database system, it can be any values stored for future reference: application data, system data (e.g. initialization files), linking data (index structures, hot lists, routing tables), or metadata. In this sense, a file of electronic mail messages that have been saved for future reference is a fair target. We call the values introduced into storage by the jammer *bogus values*. We call the values we meant to store *authentic* values. If a data item contains a bogus value, we say that *the data item has been jammed*. In order to simplify our analysis of a very complex problem, we will exclude the possibility of mistakes made by users or inadvertent flaws in software. (This does not mean that we exclude from consideration conventional data integrity techniques that also have anti-jamming properties.)

There are many possibilities open to the storage jammer. The amount and variety of stored data that is critical can be large. Other issues that significantly affect the problem include the rate and extent of the malicious changes, the method used to compute plausible bogus values, the target system architecture, and the security properties of the jamming agent. It is important to understand the nature of possible jamming attacks in order to balance the cost of defenses against the ease of making such an attack.

The main goal of this paper is to define storage jamming. We also discuss our work to date on possible defenses against it; in order to make the case that there are solutions. In the next section we discuss the nature of storage jamming including possible jamming strategies and vulnerability to jamming. The next two sections discuss a variety of anti-jamming techniques. Some of these candidate techniques are based on previous work on security-oriented data integrity but are unsuitable for use against jamming attacks. We then discuss a particular kind of anti-jamming mechanism called a detection object. The paper ends with a summary and discussion of future work.

Storage Jamming

The jammer's objective is to reduce the quality of stored data below a certain level, without being detected. Unlike conventional jamming of sensor and communication systems, we pre-

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **1995**	2. REPORT TYPE	3. DATES COVERED **00-00-1995 to 00-00-1995**
4. TITLE AND SUBTITLE **Storage Jamming**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,4555 Overlook Avenue, SW,Washington,DC,20375**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**19**	

sume that it is relatively easy to stop the jamming once it is detected[1]. If the jammer does not care if the jamming is detected then we are probably talking about a denial of service attack rather than jamming.

We describe a jammer by the conditions it uses to choose a data item to jam and the rules for determining the bogus value to be written.

Example 1

A simple random jammer might be described by the two conditions

1. jam(*storage-block*) = *storage-block* **exclusive-or** *jamming-block-value*
2. **if** there is a storage block at *disk-address*[*X*] **then** *disk-address*[*X*] := jam(*disk-address*[*X*])

where *X* is a random variable defined on the disk address space of the target system and *jamming-block-value* is a constant bit string the same size as a storage block. ❏

Example 2

A high level jammer might use conditions better described via SQL

update emp *set* sal = sal + x *where* job = ‘mathematician’;

In this case the data items are chosen by the *where* clause and the *set* clause to choose records in the *emp* table whenever *emp.job* is mathematician. The *set* clause determines the bogus value of salary to be current salary plus a random value. ❏

Jamming Strategies

It is helpful to characterize storage jamming in terms of the possible strategies. There are many possible characteristics, we consider eleven of them here:

Persistence Of Bogus Values

The unauthorized changes can be persistent or the jammer can restore the changed values after an arbitrary length of time. A useful variation of this would be to save deleted objects or values and reintroduce them at a later time. In electronic warfare terminology this would be a form of *repeat-back jamming*. Temporary bogus values may be harder to detect but may still be read by critical applications or system programs.

Security Attributes Of The Jamming Program

The jamming may be done by an authorized program or by an unauthorized program. If it is done by an authorized program it may be done as part of an authorized invocation, i.e the program simply writes incorrect values, or the jammer may be able to cause an unauthorized invocation of a legitimate application.

Target System Structure.

Target systems can vary in structure from unstructured legacy systems to modern, well-structured systems. Since poorly structured systems are hard to understand, we expect it will be harder to determine if a poorly structured system is being jammed. The modularity and encapsulation in a well-structured system isolate the effects of bogus data to a single part of the

1. We believe that there are certain kinds of mission critical legacy systems where jamming could not be stopped easily. For instance, if the system cannot be reinitialized in a convenient way, then the problem of stopping the jamming becomes much more difficult.

system. They also make it easier to determine that the source of the system error was bogus data rather than an unusual interaction of program logic.

Means Of Choosing Bogus Values

The jammer can adopt a number of basic algorithms for generating the data to write. The bogus values can be chosen arbitrarily, randomly, by interpolation, by replay, by permutation, etc. Arbitrary choices may be easy to detect, but can be performed by small programs that may be easier to insert into a system.

Means of Choosing Target Data Items

The jammer can select targets randomly, via some selection criteria, or by simply piggybacking on an application program. This last approach lets the application chose the target for the jammer. We have found it helpful to characterize the selection in terms of a condition, as in examples 1 and 2 above.

Class Of Target Data

As we said in the introduction, the data can be application data, linkage data, metadata, or system data. A more important classification of target data is the level of abstraction. For example, the units of target data (data items) could be data in a relational database or they could be disk blocks in the nodes of a B^+tree. Another important classification of target data is the size or granularity of the target data items. The jammer could target sets or lists of data, or select components of a data item, at the same level of abstraction.

Rate Of Change In Target Data

If there are many updates to the data, then jamming may be easier. There will be more opportunities and more checks will be required to find the jamming.

Rate Of Jamming

The rate at which changes are made is significant. A jammer may be designed to jam as fast as possible without being detected, with the expectation that the jammer will only be triggered at a critical moment. Alternatively, the jammer may run continuously and make changes infrequently.

The rate of jamming can be quantified, at a given level of abstraction, in terms of the number of data items jammed per state transition. One way to do this is view each high-level command as the input causing a state transition. Note that we include all state transitions, including those that only read data. For example, if we have 100 commands processed, 5 of which jam, 5 of which are authentic updates and the remaining 90 are read-only commands, then the rate of jamming is 0.5 data items per transition.

Extent Of Jamming

A slow jammer can still do much damage by using a cumulative strategy of jamming slowly but widely, i.e. ultimately change every value stored in a database. This type of jamming is usually called *barrage jamming*. On the other hand, a jammer can hope to escape detection but still disrupt operations by only modifying a critical subset of the stored data. This kind of jamming is called *spot jamming*.

The extent of jamming can be quantified, at a given level of abstraction, in terms of the number of data items jammed in a given state. For example, if we have 10000 data items in state *s* and 300 have been jammed, then the extent of jamming is 3% of state *s*. This notion of extent is dependent on the given level of abstraction and the condition used by the jammer to select targets.

Extent is an important issue for the storage jammer. If storage jamming is continuous, then at some point all of the data targeted by the jammer will be jammed. We assume that at some point before the extent of jamming reaches 100% the presence of the jammer will be detected by direct inspection by the users. For this reason, we expect the jammer will stop before such a point is reached. The jammer can then wait until normal computations change the bogus values into authentic values and then start jamming again.

Adaptability Of The Jammer

An enemy may hope to do more damage by having the jamming software changing its strategy. This may be a simple adaptation, such as changing the constraints that are checked when generating bogus values. It may be more complex, trying to adapt to detection mechanisms that might be present. As we will see later, it may be necessary to prevent the jammer from reading the data or code of the detection process, in order to frustrate this adaptability.

Means Of Introducing The Jammer

Security breaches accomplished via a network connection are currently of great interest and jamming software could certainly be introduced that way. Unfortunately, network intrusions are not the only way that jamming software could be introduced. They could be installed during software development, or installed separately after an information system is deployed. In most cases, we assume that the human who introduces the jamming software will not remain close to, or be associated with the system under attack. However, one viable strategy is to have a human there to help the jamming software. If the situation is such that a jamming agent could be reinstalled easily or assisted by a human, then the jamming would be particularly difficult to stop. This possibility may be a practical issue for large mission-critical legacy systems. These systems often cannot be shut down for maintenance and can be sensitive to small changes to seemingly unrelated data or programs. Finally, a jammer may be introduced via firmware. It would be more difficult to check firmware for malicious code than to check software that is stored in executable files.

Vulnerability to Jamming

A system's vulnerability to electronic warfare is often characterized in terms of interceptibility, accessibility, and susceptibility. *Interceptibility* is a measure of the ease with which an enemy can determine the existence, function, and location of a system. *Accessibility* is a measure of the ease with which an enemy can reach a system with an effective electronic warfare attack. *Susceptibility* is a measure of system properties that determines the effect of attacks on the system's performance. In this paper we are primarily concerned with susceptibility rather than accessibility or interceptibility.

Performance criteria for measuring susceptibility can include

1. *mission success rate*: the rate at which activities supported by the system succeed,
2. *query error rate*: the rate at which queries are not processed according to the system data model, database design, and the non-bogus portion of the system history,
3. *record error rate*: the rate at which erroneous records, object instances, etc., occur in storage,
4. *field error rate*: the rate at which erroneous fields of a record, attributes of an object,

etc., occur in storage, and

5. *bit error rate*: the rate at which erroneous bits occur in the representation of data.

Another important criteria is detection of jamming. If the jamming is detected, then we may often assume that it will cease to be effective. So a system that allows easy detection of jamming may not be very susceptible to it, even though the system has no real way of preventing or tolerating the jamming that may occur before detection.

Reducing Vulnerability to Jamming

Fortunately, there are some general software and system engineering practices that can reduce a system's vulnerability to storage jamming. Many of these practices should be followed in developing the software and firmware of critical systems.

1. The system should be well-structured. That is, it should be modular, layered, and encapsulate its data. Departures from this well-structuredness may be necessary for other reasons, but the likelihood of successful jamming is increased.
2. The system data should be designed. That is, there should be an explicit specification of the relationships between data items, their structure, and the operations that can be performed on them. Identification of data integrity constraints or invariants is a critical part of this and makes it easier to discover storage jamming that may be taking place.
3. The system behavior should be specified. This seems obvious, but is not always followed in practice. If the behavior is unspecified, then jamming is harder to detect.
4. Major state transitions of the system should be transactional. That is, they should have the transaction properties of atomicity, isolation, consistency, and durability [5].
5. Commercial off-the-shelf data management products, such as database systems, should be used for data storage instead of application-specific files. The transaction processing, encapsulation, and integrity provided by these systems makes storage jamming more difficult.
6. Fault tolerance techniques such as checksums, backup and recovery mechanisms, and redundancy may be used to increase the difficulty of jamming data.
7. Computer security techniques such as access control, audit, and identification and authentication may be used to increase the difficulty of jamming data.

These practices can reduce vulnerability to jamming, but they do not really address the problem. What we really can say about these engineering practices is that a failure to follow them may make it impossible to protect a mission critical system from storage jamming. These measures are just a starting point for dealing with the problem. A more effective way to reduce vulnerability to storage jamming is to adopt specific anti-jamming defenses.

Anti-Jamming

Defenses themselves can be either *mechanisms*: actual software or hardware constructs present in the protected system, or *measures*: practices that are followed outside the protected system, e.g. by people. An example mechanism would be a audit tool that could check integrity constraints and determine that some values in storage were not consistent[1]. An example

1. Even though verification tools are automated systems, we do not consider them mechanisms because they do not execute at the same time as the system they protect.

measure would be a blind buy of COTS to reduced the likelihood of malicious code in a critical system. The final defense is likely to be a combination of mechanisms and measures.

Anti-jamming techniques may be intended to either *prevent*, *tolerate*, or *detect* jamming. One of the most important combinations of anti-jamming measures and mechanisms is a set of backup and restore facilities that can remove the effects of jamming. Unfortunately, backup and restore facilities do not come into play until the jamming has happened. Important potential prevention mechanisms include access control and type enforcement. Both can limit the extent of the jamming by confining it within a domain, but do not completely prevent it. Since COTS applications are generally used in multiple domains, these domain-based mechanisms could allow jamming to take place in every domain that used the COTS application. Important potential toleration mechanisms include error correcting codes available from communication theory that potentially would allow a system to tolerate jamming. Unfortunately, these codes are applied to low-level data representation schemes and provide no tolerance of jamming via an agent that manipulates data at a high level of abstraction, e.g. via an SQL statement. It is not clear that these encoding schemes would extend to abstract high-level data. System development, administration, and maintenance measures that limit the introduction of malicious code are promising prevention measures, but difficult to apply in practice. Detection mechanisms and measures are less well understood, but seem to offer more promise. This promise in part stems from the fundamental nature of storage jamming: it is simple to stop the jamming once it is detected.

Anti-jamming via Partial Correctness[1]

There are several security-oriented data integrity approaches that do offer more promise: the Clark-Wilson model [2], Sandhu's transaction control expressions [12], the assured pipeline of Boebert and Kain [1], and the extended trusted path of Wiseman [16, 17]. None of these approaches, with the possible exception of the extended trusted path, was intended to deal with jamming. As we will see, they do make things harder for the jammer. Unfortunately, in systems that must use COTS or unverified software, they don't really provide significant protection against more than casual jamming attempts. This is not a shortcoming, none of these approaches was intended to defend against data jamming and, as we shall see later, jamming defenses do not necessarily provide the protection that can be achieved with these techniques. On a practical level, their purposes are orthogonal.

Clark-Wilson Well-formed Transactions

The familiar Clark-Wilson integrity model would appear to be sufficient to frustrate all but the most sophisticated of jamming attacks. But this is not the case. Implementation of an integrity policy based on well-formed transactions depends on having high-assurance trusted code (i.e. non-COTS, expensive, and difficult to produce) for every transaction. As Sandhu and Jajodia point out, in practice the trustworthiness of such transactions can vary [13]. The Clark-Wilson model is vague on what it means for every transaction in a system to be certified and on what it means for constrained data items to be protected from all other programs in a system. If less than high-assurance is provided for both the transactions and the unspecified

1. We ask the reader not to be diverted by questions regarding the nature of correctness proofs. We use the term in its broadest sense: that something much more rigorous and resource consuming than conventional software engineering is required. Because it is so easily detected, we consider nontermination to be an ineffective storage jamming technique.

mechanism that protects constrained data items from uncontrolled modification, then the jammer has relatively little difficulty in succeeding. Another problem with Clark-Wilson is that the model is designed to prevent fraud. Fraud requires a human recipient to cooperate with a Trojan horse or other malicious software. This human must be part of the system to receive assets from the system. On the other hand, jamming does not transfer assets and the enemy need not be part of the system under attack.

Both Clark and Wilson and Sandhu and Jajodia emphasize that well-formed transactions should take consistent database states to consistent database states. Although this could be sufficient for the more likely case of fraud, it is not always sufficient to prevent storage jamming. In that situation, the jamming Trojan horse can insert or update data with values that satisfy all transaction constraints, but are not authentic, i.e. they were not input to the system through its interface for the purpose of storing information about the real world. If the Trojan horse slips through a weak certification process into a Clark-Wilson transformation procedure, then its bogus data is also a valid Clark-Wilson constrained data item that would pass all checks administered by an integrity verification procedure. As Wiseman describes the problem in [17], the data is valid but not appropriate.

Example 3

We will borrow Wiseman's example of aircraft altitude [17] to make the problem clearer. Wiseman discusses an air traffic control database where civilian aircraft altitude must be an integer between 0 and 60000 feet to be a valid data state. If flight PA73's en route altitude is stored in the database as 10000 feet but PA73 is actually at 30000 feet, the data state is inappropriate since it is not the altitude we wished to have recorded, but clearly so, since commercial airlines do not operate at such low altitudes en route. Suppose that a radar sensor stores the correct altitude, 30000 feet, in the database via a valid Clark-Wilson transformation procedure, but a Trojan horse in the database system modifies PA73's altitude to be stored as 32000 feet. Air traffic controllers in a busy situation might not take any action if flight BA282, actual altitude 30000 feet, bogus altitude 28000 feet, was on a track that passed "directly under" flight PA73. They might even advise the pilots of PA73 to look for BA282 passing under them. We expect that other precautions, especially the pilots themselves, would prevent a catastrophe. However, the operations of both airlines and the air traffic control system would be degraded. ❑

Example 3 shows us that Clark-Wilson models will not prevent storage jamming unless every transformation procedure, integrity validation procedure, and underlying system procedure is proven to be partially correct. This must be so of every version of a procedure, not just the initial versions. Further measures must be applied to all software to ensure that it is not modified during maintenance or upgrades.

Well-formed Transactions Under Transaction Control Expressions

A more promising mechanism for preventing jamming is the transaction control expression of Sandhu [12]. As described, transaction control expressions encounter the same difficulties with assurance and Trojan horses as Clark-Wilson systems, but to a lesser degree. When a Clark-Wilson transformation procedure acts, it acts within a single domain. Transaction control expressions can be made to act over multiple domains. In some cases this might require the jamming Trojan horse to operate in multiple domains or more likely, multiple Trojan horses would have to cooperate. A further complication for the jammer is the human approval required by transaction control expressions. This requires more care on the part of the jamming software. We conjecture that a transaction control expression system would be harder to jam than a Clark-Wilson system, but the difficulties discussed above still remain. The most significant of these is that jamming is not necessarily directed against corporate assets that are recorded as dollar amounts (or can be mapped to dollar amounts). Jamming could be directed against map image data, object code, CAD data, sales forecasts, scheduling data, oil well logs, records of electronic signals, etc. Determining the correctness of such data by brief human inspection is problematic.

Assured Pipelines and Extended Trusted Paths

An alternative view of the integrity problem is to look at the flow of data and protect it from modification. If malicious software is unable to interpose itself into this flow, then the integrity of the data is preserved across the flow.

Assured Pipelines

Boebert and Kain proposed an integrity mechanism called an *assured pipeline* [1]. An assured pipeline is a transforming subsystem that must be encountered by a specific data flow. The examples given concern labeling or encryption of data just prior to output. An assured pipeline has three properties:

1. The security-relevant subsystem cannot be bypassed.
2. Transforms made by the pipeline cannot be undone or modified.
3. Transforms made by the pipeline itself are correct.

The example given for a trusted pipeline is protection of output. This is a special case that does not flow data into a store. Stored data is not protected by an assured pipeline, only data under processing. One difficulty is that property 2 is very difficult to ensure for stored data, particularly when the data is shared. Other measures or mechanisms are required to prevent the stored data from being modified by software outside the pipeline. Once again, we find that assured pipelines are only effective if all of the software that might be able to modify stored data is shown to be partially correct.

We also encounter difficulty when the components of the assured pipeline are large and complex, e.g. database system software. This makes it difficult to achieve property 3 of an assured pipeline.

Extended Trusted Paths

Wiseman [17] proposed the use of an extended trusted path, as exemplified by the SMITE system [16]. The SMITE trusted path is "all [of the] software which needs to be executed in order for the wishes of the human user to be invoked." Software on the extended trusted path is shown to be totally correct (and presumably kept that way). If we could support the extended trusted path with some kind of strong fine-grained domain protection, such as type enforcement, we could exclude all sources of jamming via the trusted path. In systems where this is possible, we think jamming would be extremely difficult.

Alternatives to Partial Correctness

The point of the preceding discussion is that four different potential anti-jamming mechanisms lead us to the same condition, namely that we must expect everything to be partially correct. But commercial off-the-shelf software or low-assurance customized applications are the order of the day. Older (legacy) systems are perhaps very-low-assurance systems because of their poor structure and thus even less likely to be protected by any of the mechanisms discussed so far. We conjecture that partial correctness will never be provided for the entire path followed by information from input to output, for the entire life of a system. Instead, at best we can assume that some small subsystem is sufficiently correct to defend against jamming.

As we have already shown, reference monitors that check accesses are not very effective against storage jamming. A successful mechanism will probably not be based on directly on cryptography either. Extensive use of cryptography would defeat the purpose of many internal data structures, and might not be feasible with off-the-shelf applications. Low level encryption engines would not provide protection against jamming at a high level of abstraction, e.g. via the methods of an object-oriented application program.

One alternative that we will leave to another paper is the possibility of having multiple versions of the software developed by separate teams. Each version of the software would vote for or compare the values it intended to store. In principle, the trusted parts of such a mechanism would be a trusted input multiplexer, a trusted comparison or voting mechanism, and a trusted output demultiplexer. Although this kind of n-version programming has been shown to be of limited use in preventing unintentional flaws, it would be a different matter to use it to prevent misbehavior via malicious software. Presumably, the likelihood of compromising two or more development teams in a way that allows coordination between the malicious components is very small.

Anti-jamming via Detection Mechanisms

The alternative mechanisms we propose here would be background systems that detect jamming in a timely fashion. The alternatives are based on data architecture; the strategy is to arrange the data storage in such a way that jamming changes are easily detected. The three mechanisms we have identified so far are specialized integrity constraints, multi-process multi-domain transactions, and detection objects.

Specialized data integrity constraints can simplify detection because the detection software could check them efficiently but the jammer would have some difficulty in computing plausible bogus values that satisfied them. *Multi-process multi-domain transactions* extend this concept by structuring updates, deletes, etc. in such a way that no single process could determine plausible bogus values. Finally, *detection objects* are data structures that appear to be part of an application, but are not used. If these objects are changed, then there is a high probability that the change was made by a jammer.

Detection Objects

A detection object is an abstract mechanism that is intended to detect the actions of malicious software that jams storage. It overcomes the difficulty of checking the computation performed by a program by always remaining in a predictable state. If the detection object is not in its proper, predictable state, then it was probably modified by a jammer. We call the data items that are intended to store legitimate data, i.e. not detection objects, *protected data items*. Protected data items and detection objects are defined in terms of jammers that might target them. Detection objects satisfy two properties

1. Detection objects are indistinguishable to the jammer from their corresponding protected data items, that is, they satisfy the same condition as the jammer. Call this condition *indistinguishability*.
2. The only legitimate process that modifies detection objects is the jamming detection process. Call this condition *sensitivity*.

Example 4

If we wished to protect the aircraft altitude records of Example 3, we could insert records for flights that did not exist, for example flight QQ99 altitude 28000. The detection process would have internal storage that it could compare and test for the presence of the QQ99 record and its “correct” altitude of 28000. Since the altitude value is precomputed, and no radar sensor report will ever make an authentic update to QQ99, any change to it must be a bogus update. ❑

If a detection process inserts enough detection objects into the storage structures of an information system, an active jammer will eventually jam one of the detection objects and be detected. A simple model of this can be constructed if we assume that the jammer is equally likely to jam any of the potential targets, protected data items or detection objects. Suppose we include enough detection objects to make them be 1% of the total storage. This gives a probability p of jamming a protection object as $p = 0.01$. We can then represent the continued jamming as a binomial distribution with n jamming attempts as trials and the i successes as jamming of detection objects. We have the binomial distribution function

$$\mathrm{P}\{X \leq i\} = \sum_{0 \leq k \leq i} \binom{n}{k} p^k (1-p)^{n-k}$$

with X being the number of jamming changes made to detection objects. With 1% of the total storage being detection objects, the probability of detecting the jammer after 300 changes is 0.95, and 0.63 after only 100 changes. If the percentage of detection objects is raised to 2%, then the probability of detection becomes 0.95 after only 150 changes are made. From this we see that a moderately sized set of data, say 10,000 storage objects, is efficiently protected by detection objects. With 2% of the members as detection objects, our example data set has 9800 protected data items and we have a 95% chance of detecting the jammer after less than 150 of the protected data items have been jammed[1]. On the other hand, if we have only 100 objects in the data set then the entire set could be compromised with only a 63% chance that the detection objects would reveal the jamming. The answer in this case (small sets of data) is to use a much higher proportion of detection objects. If we add 100 detection objects to protect a set of 100 objects, then the probability of jamming a detection object is one-half. This gives us a 97% chance of detecting the jamming after only five data items have been changed. The reduced efficiency is tolerable for smaller sets.

Detection objects only protect the sets of data to which they correspond. For example, if there are two databases *aircraft tracks* and *flight plans* on a system, with detection objects inserted into database *aircraft tracks* but not database *flight plan*, and a jammer only modifies database *flight plan*, then the jamming will not be detected. The exact meaning of “correspond” is not straightforward. It is difficult to assert that a given set of protection objects “protects” a specific set. We might say that a detection object y protects a set P of data items if y is a member of P. However, the jammer might only be designed to select a subset of P. If we call J the set of all data items that could potentially be modified by a given jammer j and D the set of all detection objects, then the set D_j is the set of all detection objects that are also in J, that is

1. In more complex models, we also need to allow for latency in the detection process. A change to a detection object may not be discovered as soon as the jammer makes the change.

$D_j = \{ x \mid x \in J \text{ and } x \in D \}$; in our binomial distribution example D_j contained 200 detection objects. Likewise P_j is the set of all protected data items that are in J, that is, $P_j = \{ x \mid x \in J \text{ and } x \notin D \}$; in our example P_j contained 9800 protected data items. If D_j is nonempty then we can say that D_j protects P_j against jammer j. So we see that detection objects protect data items against specific jammers that target both those data items and some of the detection objects. The problem of exactly specifying the set of all jammers that might be detected by a given set of detection objects is a difficult one and beyond the goals of this paper. (It is related to the set cover problem which is NP-complete.)

Preserving Indistinguishability and Sensitivity

Our discussion up to this point has little significance unless we can implement detection objects that preserve indistinguishability and sensitivity. We can implement plausibly indistinguishable detection objects by using the types, schema, class definitions, etc. of the data items we intend to protect. We can further preserve indistinguishability by storing plausible values in the detection objects. The conditions that a jammer may select on are not limited to stored values from a single system state. Other attributes may be used: for example time of last update, a pointer value in the representation of the data (e.g. jam all linked list entries where the link value is odd), and data item names (e.g. jam all files whose names begin with the letter Q). The jammer may also look at more than one state, in order to jam only those data items that change rapidly. The closest approximation to complete indistinguishability results when we create detection objects with values of every attribute chosen using the same distribution as the corresponding values in the protected objects. [1] In practical cases, we can estimate the distributions, since practical jammers that could detect flaws in our estimated distributions would be both difficult to write and also relatively large programs. We consider this latter characteristic to be undesirable (from the malefactor's perspective) in malicious software because it significantly increases the likelihood that the malicious software will be detected for some reason other than modifying a detection object.

Preserving sensitivity can be difficult and can conflict with indistinguishability. If applications are to leave detection objects alone, then there must be some means for the applications to distinguish the detection objects. There are two risks here: first, if such an attribute is in use, the jammer may be able to discover it; second, the jammer may be a Trojan horse in an application and may use the application logic to bypass the detection objects, without discovering what the distinguishing attribute is.

Commingled-Object Detection

One alternative strategy for preserving sensitivity is not to have a distinguishing attribute, that is, only the detection process will be able to determine if a given data item is in fact a detection object. With this approach, the detection process installs the detection objects in the data structures of the protected system and records some attribute of the installed object (such as an address) that the detection process can use to identify it later as a detection object. Under these circumstances, a program would change the detection objects if those objects were accessed by the program. To effectively scan for jamming, the detection process would first reset all of the detection objects to the proper state and then run the programs to be scanned. The scanned programs would be run using a script which should cause the programs

1. This is the reason for defining sensitivity to include modification of detection objects by the detection software. It may be necessary to give the detection objects a plausible history.

to set the detection objects to another proper state. If the some of the detection objects are not in the expected state, then there may be jamming. We call this strategy *commingled-object detection* because it intersperses detection objects with protected data items.

If detection objects are commingled with application or system data, their contents may be returned as part of a query. To prevent this, the detection process can provide a service or call that identifies detection objects, such as *isDetectionObject(x)* so that they need not be displayed. We need to restrict this service so that a jammer cannot use it to escape detection. Since we only try to detect jamming by having the detection process invoke a program against a predefined script, use of the identification service can be restricted to processes that are not being invoked to detect jamming. If a process has been invoked for the purpose of scanning it for jammers, then the detection process can always return false when asked if an object is a detection object.

Example 5

Suppose we wish to protect an air traffic control system from jamming by using commingled objects. Our protected system has two databases: *aircraft-tracks* and *flight-plans,* as shown below. Detection objects are shaded; there are two of them in this example:

Flight-Plans

Flight	Origin	Destination	ETA
BA282	LHR	IAD	0935
PA73	MSY	FRA	1410
TW37	LHR	JFK	0940
QQ99	FRA	BWI	1105

Aircraft-Tracks

Flight	Course	Speed	Altitude
BA282	290	1250	40000
PA73	075	625	37000
TW37	285	595	34000
QQ99	085	595	33000

The detection process could identify the detection objects in *flight-plans* and *flight-tracks* as the fourth object in each table. To check or scan the software that manages *flight-plans*, the detection process could execute an update that set *flight-plans.eta* to 1110 where *flight-plans.flight* was QQ99. Before executing this test command via a script, the detection process would set the fourth record in *flight-plans* to the known state shown above and also set the fourth record in *aircraft-tracks* to the known state shown above. The detection process would then run the command to update QQ99 in *flight-plans*. After running the command, it would check both the fourth record in *flight-plans* and the fourth record in *flight-tracks*. If the altitude for QQ99 was 37000 instead of 33000, the detection process would indicate probable jamming to the system security officer. If the jamming software changed the altitude of TW37, this would not be picked up by the detection process. Also, if the altitude of QQ99 was changed by a radar sensor update, the detection process would either have to resolve the change or raise a false alarm. ❏

Commingled-object detection is suitable for protecting data that does not change frequently. If there are lots of updates taking place, these updates may interfere with the scanning. A detection object may be changed by a program that is not being scanned. This interference would require the detection process to log and monitor all updates in order to resolve possible problems. Commingled-object detection is also less effective against slow jammers that may decide not to jam on every program invocation.

Quarantine Approaches

Another alternative for preserving sensitivity is to have the distinguishing attribute be implicit and not available to potential jammers. This strategy can be carried out in three ways: *quarantine systems*, *quarantine subsystems*, and *quarantine applications*. The quarantine system approach is the easiest to explain and also the most powerful detection mechanism. A quarantine system is a copy of the system to be protected. It has all of the programs that run on the protected system. All of the data in the quarantine system is put there by a script that leaves the quarantine system in a predetermined state at all times. The script uses exactly the same data definitions, metadata, class definitions, etc. as the system to be protected. No users are accessing the quarantine system and it is not connected to any operational system. The script can be designed to loop continuously so that the quarantine system is always active and under a plausible load. All of the data in the system can be checked quickly by the background detection process because the data is in a predetermined state that is known to the detection process. In a quarantine system, the set of protected data items P_j is empty and the set of potentially modified detection objects D_j has 100% of the data items in J, for any jammer j. The quarantine system protects any operational system that has precisely the same software installed. To the extent that the operational system has other software installed, it is unprotected. It might be possible for a jammer to avoid detection by only activating itself on certain installation specific parameters such as memory size or number of I/O buffers configured. To allow for this, the script can be run against the quarantine system under an appropriate variety of installation parameters.

A quarantine system does not have to be at the same location as the system it protects. A complete copy of a new or upgraded system can be set up and run before copies of it are sent out to operational sites. After the copies of the system are running at the operational site, the quarantine system can continue to run in parallel, to protect against jammers that use a delay mechanism. This approach could also be used to save time, by doing some quarantine processing before operational use, but completing the processing while systems were in operation. The quarantine system does not have to be as large or complete as the operational systems it protects. It only needs to be able to run all of the software under the same installation parameters as the protected systems.

Example 6

Suppose we set up a quarantine system to protect our air traffic control system. On a separate hardware platform, we install the air traffic control software, with simulated inputs controlled by the detection process. Our two databases have the same format as before but every record in the database is a detection object.

Flight-Plans

Flight	Origin	Destination	ETA
QQ37	LHR	JFK	0940
QQ99	FRA	BWI	1105

Aircraft-Tracks

Flight	Course	Speed	Altitude
QQ37	285	595	34000
QQ99	085	595	33000

Now the detection process can run a command that should set *flight-plans.eta* to 1110 where *flight-plans.flight* was QQ99. There is no need to preset any data and there is no possibility of interference from an application or system process. Also, the detection process is much simpler because it does not need to distinguish detection objects from protected data items. After the update, the detection process will be able to detect any bogus change to any part of each table. For instance, if the jammer swaps *aircraft-tracks.flight* values while it is supposed to be updating *flight-plans*, then it will be detected. ❑

Quarantine systems will detect slow jammers, random bit-level barrage jammers, spot jammers, programs that jam by changing data outside their own application, and programs that jam by deliberately writing incorrect values.

A quarantine subsystem is like a quarantine system that runs on the same hardware as the operational system it protects. The quarantine subsystem is set up as a protected subsystem [4] that runs the same software and uses the same data definitions, metadata, etc. as the protected system. Predefined scripts exercise the software and always leave the data in a predetermined state. Once again, the set P_j is empty and the set D_j has 100% of the data items in J, for any jammer j. In this case, the operational software is protected to the extent that potential jammers are unable to determine that they are currently executing against the quarantine subsystem. The benefit of a quarantine subsystem is that it allows each site to have different software installed. The disadvantage is that the features of the operational system must be extended to provide the quarantine subsystem.

Example 7

Suppose we decide to protect our air traffic control system with a quarantine subsystem. Now we have four databases, as shown below:

Flight-Plans-1

Flight	Origin	Destination	ETA
QQ37	LHR	JFK	0940
QQ99	FRA	BWI	1105

Aircraft-Tracks-1

Flight	Course	Speed	Altitude
QQ37	285	595	34000
QQ99	085	595	33000

Flight-Plans-2

Flight	Origin	Destination	ETA
BA282	LHR	IAD	0935
PA73	MSY	FRA	1410
TW37	LHR	JFK	0940

Aircraft-Tracks-2

Flight	Course	Speed	Altitude
BA282	290	1250	40000
PA73	075	625	37000
TW37	285	595	34000

In this case, *flight-plans-1* and *aircraft-tracks-1* form the quarantine subsystem. The databases *flight-plans-2* and *aircraft-tracks-2* form the operational air traffic control system. All four tables are on the same platform. The detection process exercises the operational software against *flight-plans-1* and *aircraft-tracks-1*. All of the values in *flight-plans-1* and *aircraft-tracks-1* are put there by the detection process, so the detection process is as simple as the quarantine subsystem but able to detect the same wide range of jamming behaviors. ❑

A quarantine application is like a partial quarantine subsystem. Instead of using all of the programs and data definitions of the operational system, a quarantine application runs a script against the programs, data definitions, metadata, etc. of a single application.

Example 8

Now we will protect our air traffic control system with a quarantine application. In this case the only detection objects are those that appear to be aircraft-tracks data items:

Aircraft-Tracks-1

Flight	Course	Speed	Altitude
QQ37	285	595	34000
QQ99	085	595	33000

Flight-Plans

Flight	Origin	Destination	ETA
BA282	LHR	IAD	0935
PA73	MSY	FRA	1410
TW37	LHR	JFK	0940

Aircraft-Tracks-2

Flight	Course	Speed	Altitude
BA282	290	1250	40000
PA73	075	625	37000
TW37	285	595	34000

In this case the detection process only runs commands to modify the table *aircraft-tracks-1*. It also can periodically check *aircraft-tracks-1* without running a predefined command, in order to catch jamming against *aircraft-tracks* via a *flight-plans* application. This is also a good example of how important it is to describe jamming detection in terms of specific jammers. Even though this quarantine application is based on *aircraft-tracks*, it will also detect low-level barrage jammers that happen to corrupt part of the high-level *aircraft-tracks-1* detection objects.❑

Detection Objects in the System Life Cycle

The efficacy of a detection object defense is significantly influenced by the way it is included in the system life cycle. Some parts of a detection object defense should be deferred until late in the development process, or even until after the system has been installed. Choice of the actual detection objects and the scripts used by the detection process should be deferred until the system is in its final form and no new code is likely to be introduced. It might be preferable to have the detection objects and the corresponding scripts developed by the system security officer, or by a separate development team. The critical point is that the indistinguishability of the detection objects depends on the assumption that the person designing the jamming software does not know the specification of the detection objects, or the scripts that will be run against them. It should be easier to achieve this if the specific detection object design is done after the software and firmware for a system has been put in place.

On the other hand, the background detection process should have an open design that is carefully integrated with the system it protects. It should be a high-assurance component that is shown to scan correctly, protect itself from tampering, and protect the identity of the detection objects and their associated scripts. The ability to protect itself from tampering must be relatively high assurance, since it is intended to function in a an environment where random modification of data is taking place. The best means for doing this is probably something like an assured pipeline or extended trusted path. In any case, the background detection process should be included in the early phases of the system life cycle.

Scanning Detection Objects

The overall efficiency of a detection-object defense depends on not only careful management of storage for detection objects but also on low overhead for the scanning process. It should be possible to compare the actual value of the detection object with the expected value stored by the background detection process with relatively few machine cycles.

The granularity of the checks is significant. If we are only interested in detecting a change to our current set of detection objects, then a simple checksum over the entire set via a polynomial code will suffice. We probably want to do more than that. Locating the precise changes made by a jammer will help us to locate not only the jammer but any possible damage it may have done to our operational data. Therefore, we want to be able to compare individual detection objects with their expected values. Unfortunately, to do this we have to store the expected bit-level representation of each detection object, along with any access structures it may be embedded in. This problem becomes even harder when we maintain a plausible history for our detection objects. The size of the store we need to maintain is a function of the length of the cycle we want our history to run through; if we have *d* detection objects of size 1 and we want to change the state of all detection objects twice in our plausible history, then we need a minimum of 3*d* units of storage to be able to compare each detection object to its expected value. We can mitigate this problem by using data compression and off-line storage. We can also develop a hierarchy of checksums. Each set of detection objects can be partitioned, with checksums for each block of the partition. This partitioning can be repeated until the desired granularity of checking is reached.

Large-granularity detection objects need fewer distinguishing attributes to maintain sensitivity; sets of detection objects look more like the real thing. This is one of the merits of the quarantine systems approach over the commingle-object approach. It is easier to construct an initial checksum that covers all detection objects.

Scripts that are used for jamming detection must satisfy the same properties as the detection objects. The commands in the scripts must be indistinguishable from plausible application or system commands that might be run against the data items to be protected. The scripts must, for each state transition, leave all of the detection objects in the predefined state expected by the detection process. Additionally, the commands of the script must not allow other processes to modify detection objects. Finally, the scripts themselves must be inaccessible to jammers.

Assurance

An effective detection object system should have assurance that it satisfies the two properties of indistinguishability and sensitivity. It should also have assurance that the protected subsystem cannot be tampered with or observed. In the latter case, a jammer could inspect the scripts so as to avoid jamming against the detection system's scanning. The overall assurance we have that jamming will be detected is no more than the assurance we have in these two properties.

Summary

The real-world operations of modern organizations can be disrupted by storage jamming of their supporting information systems. The object of the jammer is to reduce data quality without being discovered and there are many ways to accomplish this.

Storage jammers can be described in terms of the condition they use to select data items to jam and the condition they use to chose bogus values. This paper has shown eleven aspects of storage jamming strategy. A system's vulnerability to storage jamming may be measured in terms of interceptibility, accessibility, and susceptibility. Our primary concern has been to reduce susceptibility. Susceptibility can be reduced by first following certain general system engineering practices and then adopting specific anti-jamming techniques. There are several security-oriented data integrity approaches that do have anti-jamming properties, but they all depend on showing partial correctness in all application and system software. In this paper we show how only a small subsystem needs to be trusted in order to provide significant anti-jamming protection. The most promising mechanism is a background detection process that can detect jamming in a timely fashion because data has been organized to allow this. Three possible data organization approaches are specialized data integrity constraints, multi-process multi-domain transactions, and detection objects.

A detection object is an abstract mechanism that overcomes the difficulty of checking computation by always remaining in a predictable state. If a detection object is not in its expected state, then jamming is probably taking place. Detection objects must satisfy two properties: indistinguishability (jammers cannot distinguish detection objects from other data) and sensitivity (high probability that an unexpected detection object state indicates jamming). Possible implementations of detection objects include the commingled-object, quarantine system, quarantine subsystem, and quarantine application approaches. Design and implementation of detection objects and their associated scripts should be deferred until late in a system's life cycle. On the other hand, the design and integration of the background detection process should be started as early as possible in a system's life cycle. A detection object system should be able to quickly scan its detection objects, a function best performed by a checksum computed over many objects. Additional scanning techniques should be used to allow the detection object system to pinpoint the jammed data.

It is not clear that our proposed anti-jamming mechanisms would be effective in preventing fraud. Fraud is most likely to be carried out by causing improper sequences of correct commands, with an eye to moving assets outside the system. These sequences are improper because the humans initiating them are initiating them with improper input. Furthermore, the human who wishes to receive the assets diverted by computer fraud does not want degraded system operation, but rather correct and efficient allocation of resources to the wrong destination. It may be possible to define certain kinds of fraud that would be detected by our mechanism, but we believe that more specific measures that enforce separation of duties on users or roles are more appropriate. In many cases, the jamming detection software would not detect the fraud because the improper sequences of commands might not do anything to a detection object.

The issues we have discussed here are more complex than we have had room to discuss. Our future work will address them separately but in more detail. There are three necessary elements in future work. One is to put on a black hat and learn how systems may be jammed, in order to help identify the types of threats and the severity of the threats that are likely to be encountered. The second is to study a variety of techniques to reduce the risks due to jamming. These techniques include the detection objects introduced here, and approaches from other disciplines: Information dispersal algorithms, predicate detection algorithms, and snapshot algorithms for distributed systems. Third, we would like to be able to provide strong jamming protection for systems that otherwise have only weak security features. Many anti-jamming techniques require a protected subsystem. How does one build a protected subsystem using relatively weak underlying security features? How strong does the protected subsystem have to be to effectively support a given level of anti-jamming?

References

1. BOEBERT, W.E. and KAIN, R.Y. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference* (Gaithersburg, Maryland, 1985). 18-28.
2. CLARK, D.D. and WILSON, D.R. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, April 1987). 184-194.
3. DEFENSE SCIENCE BOARD. *Report of the Summer Study Task Force on Information Architecture for the Battlefield*, December 20, 1994.
4. GASSER, M. *Building a Secure Computer System*. Van Nostrand Reinhold, 1988.
5. GRAY, J. and REUTER, A. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman, 1993.
6. HINKE, T. DBMS technology vs. threats. In *Database Security: Status and Prospects*, ed. C. Landwehr, North-Holland, Amsterdam, 1988, 57-87.
7. JUNEMAN, R.R. Integrity controls for commercial and military applications, II. In *Report of the Invitational Workshop on Data Integrity* (RUTHBERG, Z.G. and POLK, W.T. editors), NIST, Special Publication 500-168 (September 1989).
8. KATZKE, S.W. and RUTHBERG, Z.G. (editors). *Report of the Invitational Workshop on Integrity Policy in Computer Information Systems (WIPICS)*, NIST, Special Publication 500-160, (January 1989).
9. RUTHBERG, Z.G. and POLK, W.T. (editors). *Report of the Invitational Workshop on Data Integrity*, NIST, Special Publication 500-168 (September 1989).
10. SANDHU, R.S. The schematic protection model: its definition and analysis for acyclic attenuating schemes. *JACM 35*, 2 (April 1988). 404-432.
11. SANDHU, R.S. Terminology, criteria and system architectures for data integrity. In *Report of the Invitational Workshop on Data Integrity* (RUTHBERG, Z.G. and POLK, W.T. editors), NIST, Special Publication 500-168 (September 1989)
12. SANDHU, R.S. Separation of duties in computerized information systems. In *Database Security IV: Status and Prospects* (JAJODIA, S. and LANDWEHR. C.E., editors). North-Holland 1991, 179-189.
13. SANDHU, R.S. and JAJODIA, S. Integrity mechanisms in database management systems. In *Proceedings of the 13th NIST-NCSC National Computer Security Conference* (Washington, DC, October 1990), 526-540.
14. THOMSEN, D.J. and HAIGH, J.T. A comparison of type enforcement and Unix setuid implementation of well-formed transactions. In *Proceedings of Sixth Annual Computer Security Applications Conference* (Tucson, Arizona, December 1990), 304-312.
15. ULLMAN, J.D. *Principles of Database and Knowledge-Base Systems*, Vol. 1. Computer Science Press, 1988.

16. WISEMAN, S., TERRY, P., WOOD, A., and HARROLD, C. The trusted path between SMITE and the user. In *Proceedings of the IEEE Symposium on Security and Privacy* (Oakland, California, April 1988). 147-155.

17. WISEMAN, S. The control of integrity in databases. In *Database Security IV: Status and Prospects*, (JAJODIA, S. and LANDWEHR. C.E., editors).North-Holland 1991, 191-203.